Effcient storage of Pareto solutions in biobjective mixed integer programming

http://dx.doi.org/10.1287/ics.2015.XXXX

14th INFORMS Computing Society Conference
Richmond, Virginia, January 11–13, 2015
pp. 000–000

Efficient storage of Pareto solutions in biobjective mixed integer programming

***Nathan Adelgren*[†]*, Pietro Belotti*[‡]*, and Akshay Gupte*[†]**

[†]Department of Mathematical Sciences, Clemson University, Clemson, SC 29634,
nadelgr@clemson.edu, agupte@clemson.edu

[‡]FICO, International Square, Starley Way, Birmingham B37 7GN, United Kingdom,
pietrobelotti@fico.com

Abstract Many of the techniques for solving biobjective mixed integer linear programs (BOMILP) are iterative processes which utilize solutions discovered during early iterations to aid in the discovery of improved solutions during later iterations. Thus, it is highly desirable to efficiently store the nondominated subset of a given set of solutions. To this end, we present a new data structure in the form of a modified binary tree. The structure takes points and line segments as input and stores the nondominated subset of the input. We perform two computational experiments. The results of the first show that this structure processes inserted data faster than alternative structures currently implemented in the literature. Results of the second experiment show that when our structure is utilized inside fathoming procedures for biobjective branch-and-bound (BB), the running times for BB are reduced in most cases.

Keywords biobjective mixed integer, Pareto set, quad-tree, data structure, fathoming.

1. Introduction

Biobjective mixed integer linear programs (BOMILP) have the following form,

$$\begin{array}{ll} \min_{x,y} & f(x,y) := \left[f_1(x,y) := c_1^\top x + d_1^\top y,\ f_2(x,y) := c_2^\top x + d_2^\top y\right] \\ \text{s.t.} & (x,y) \in P_I := \{(x,y) \in \mathbb{R}^m \times \mathbb{Z}^n :\ Ax + By \leq b\} \end{array} \tag{1}$$

where P_I is a bounded set. Define $\Omega := \{\omega \in \mathbb{R}^2 : \omega = f(x,y)\ \forall (x,y) \in P_I\}$ the collection of all points in $\mathbb{R}^2$ which can be obtained using the objective function values of feasible solutions to (1). We refer to the space $\mathbb{R}^2$ containing Ω as the *objective space*.

Unlike single-objective programs, one cannot expect to find a single optimal solution to biobjective programs since the objective functions are often conflicting. Instead, a set of solutions which offer an acceptable compromise between the objectives is sought. In order to determine which solutions are "acceptable," we provide several notations and definitions. For two vectors $v^1, v^2 \in \mathbb{R}^2$ we use the following notation: $v^1 \leqq v^2$ if $v^1_i \leq v^2_i$ for $i = 1, 2$; $v^1 \leq v^2$ if $v^1 \leqq v^2$ and $v^1 \neq v^2$; and $v^1 < v^2$ if $v^1_i < v^2_i$ for $i = 1, 2$. Given distinct $(\overline{x}, \overline{y}), (x', y') \in P_I$, we say that $f(\overline{x}, \overline{y})$ *dominates* $f(x', y')$ if $f(\overline{x}, \overline{y}) \leq f(x', y')$. This dominance is *strong* if $f(\overline{x}, \overline{y}) < f(x', y')$; otherwise it is *weak*. A point $(\overline{x}, \overline{y}) \in P_I$ is (*weakly*) *efficient* if $\nexists\ (x', y') \in P_I$ such that $f(x', y')$ (strongly) dominates $f(\overline{x}, \overline{y})$. The set of all efficient solutions in P_I is denoted by X_E. A point $\overline{\omega} = f(\overline{x}, \overline{y})$ is called *Pareto optimal* if and only if $(\overline{x}, \overline{y}) \in X_E$. Given $\Omega' \subseteq \Omega$ we say that $\omega' \in \Omega'$ is *nondominated* in Ω' if $\nexists\ \omega'' \in \Omega'$ such that ω'' dominates ω'. Note that Pareto optimal points are nondominated in P_I. A BOMILP is considered solved when the set of Pareto optimal points $\Omega_P := \{\omega \in \mathbb{R}^2 : \omega = f(x,y)\ \forall (x,y) \in X_E\}$ is found.

Let $Y = \text{Proj}_y P_I$ be the set of integer feasible subvectors to (1). Since P_I is bounded, we have $Y = \{y^1, \ldots, y^k\}$ for some finite k. Then for each $y^i \in Y$ there is an associated BOLP, referred to as a *slice problem* and denoted $\mathbb{P}(y^i)$, obtained by fixing $y = y^i$ in (1). Problem $\mathbb{P}(y^i)$ has a set of Pareto solutions $S_i := \{(f_1, f_2) \in \mathbb{R}^2 : f_2 = \psi_i(f_1)\}$, where $\psi_i(\cdot)$ is a continuous convex piecewise linear function. Then $\Omega_P \subseteq \cup_{i=1}^k S_i$ and this inclusion is strict in general. In particular, we have $\Omega_P = \cup_{i=1}^k \left(S_i \setminus \cup_{j \neq i} \left(S_j + \mathbb{R}^2_+ \setminus \{\mathbf{0}\}\right)\right)$. Such union of sets is not, in general, represented by a convex piecewise linear function. It should be noted that finding Ω_P is not a trivial task in general. In the worst case, $\Omega_P = \cup_{i=1}^k S_i$ and one may have to solve every slice problem to termination, which can have exponential complexity. For multiobjective IP's (i.e. $m = 0$), De Loera et al. [3] prove that Ω_P can be enumerated in polynomial-time for fixed n, which extends the well known result that single-objective IP's can be solved in polynomial-time for fixed n. We are unaware of any similar results for BOMILP.

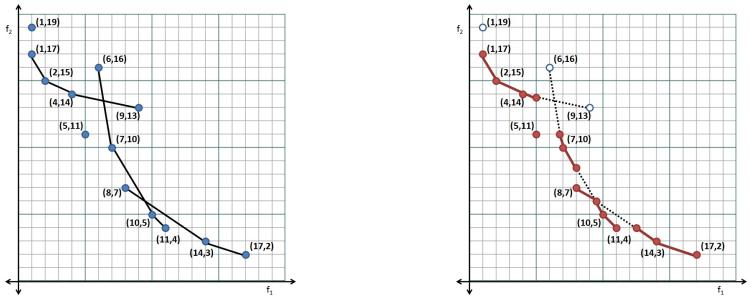
The works of Belotti et al. [1] and Boland et al. [2] are the only exact procedures that we know of for solving BOMILP with general integers, though Özpeynirci and Köksalan [9] give an exact method for finding supported solutions of BOMILP. There are also other techniques in the literature which have been devoted to specific cases. Many of the solution methods for BOMILP are based on biobjective branch-and-bound (BB) procedures [1, 7, 14, 17], but other techniques have also been used [2, 9]. We also note that the pure integer case has been studied for binary variables [6], general integers [11] and specific classes of biobjective combinatorial problems [5, 10, 13].

In this paper we present a data structure for efficiently storing a nondominated subset of feasible solutions to a BOMILP. This structure is useful alongside exact solution procedures as well as heuristics which aim to approximate the Pareto set. The structure we present is a modified version of a *quad-tree*. For a detailed description and background information on quad-trees, we suggest [12]. Although quad-trees have been used extensively for storing Pareto points in the past [15, 16], they have been used only in the pure integer case. However, notice that in the pure integer case all nondominated solutions are singletons while in the mixed integer case nondominated solutions can consist of line segments as well as singletons. Therefore, since we consider the mixed integer case in this work, our data structure stores line segments as well as singletons. Data stored in a quad-tree is organized in such a way that it can be easily searched to find a desired subset, which is desirable in certain situations. The algorithms we use to implement this tree force it to remain balanced, because having a balanced tree reduces the worst case time complexity required to access an individual node.

In Section 2 we describe the structure in detail, provide the algorithms necessary for its implementation, and discuss the complexity and correctness of each algorithm. Section 3 provides an example of using the structure to determine the nondominated subset of a given set of solutions. In Section 4 we conduct two experiments. The results of the first experiment show that in the mixed integer case our data structure is able to store nondominated solutions more efficiently than a dynamic list and can handle the insertion of up to 10^7 solutions in reasonable time. In the second experiment we utilize our structure alongside the BB procedure of Belotti et al. [1] to solve specific instances of BOMILP. The results show that the use of our structure leads to faster solution times for almost all solved instances.

2. Tree data structure

We begin this section by presenting the high-level idea of our data structure. Next we give a detailed description of the data structure and the algorithms we used to implement it. We finish by discussing some theoretical results including the complexity of each algorithm, and thus the overall structure. Throughout this discussion, when we refer to storing solutions we are referring to points in the objective space. Recall that we will be storing the nondominated subset of the union of several Pareto sets. One convenient way to store this subset is to store each of the individual points and line segments in $\mathbb{R}^2$ that it comprises.



(a)Generated solutions (b)Nondominated subset

FIGURE 1. Example of solutions generated during the solution of an instance of BOMILP.

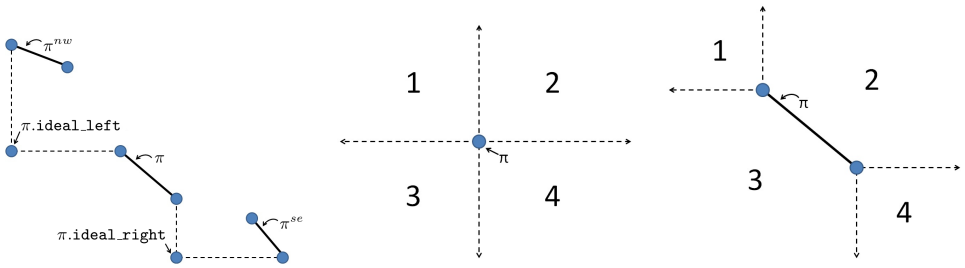
2.1. Purpose and principle

Figure 1(a) shows an example of solutions that might be generated when solving an instance of BOMILP. We would like to store the nondominated portion of these points and segments, as shown in Figure 1(b). Our goal is to have a data structure $\mathcal{S}$ which can take points and line segments as input, and store only the nondominated subset of the input regardless of the order in which solutions are inserted. Therefore, when a new solution is added to $\mathcal{S}$, it needs to not only recognize whether or not this new solution is dominated by solutions already in $\mathcal{S}$, but it must also be able to determine whether or not the new solution dominates any currently stored solutions. Once these checks have been made, $\mathcal{S}$ must be able to update so that it contains only nondominated solutions. Consider the solutions shown in Figures 1(a) and 1(b) and suppose that the segments connecting (1,17), (2,15), (4,14), and (9,13) are in $\mathcal{S}$. When inserting the point (5,11), $\mathcal{S}$ must recognize that (5,11) dominates a portion of the segment connecting (4,14) and (9,13), and thus this portion of the segment must be removed from $\mathcal{S}$ before (5,11) is added. Similarly, when the segment connecting (6,16) and (7,10) is inserted, $\mathcal{S}$ must recognize that a portion of this segment is dominated by (5,11) and only allow the nondominated portion of the segment to be added. The data structure we use is a modified version of a quad-tree in which each node represents either a singleton or a line segment associated with a Pareto point or set of Pareto points of (1). Note that a quad-tree is a data structure specifically designed for storing data in $\mathbb{R}^2$. Each node π in a quad-tree has at most four children, one for each each quadrant of the Cartesian plane. The four children of π must lie within $\pi+\mathbb{R}_{++}, \pi+\mathbb{R}_{-+}, \pi+\mathbb{R}_{--}$ and $\pi+\mathbb{R}_{+-}$, respectively, where, for example, $\mathbb{R}_{++} := \{x \in \mathbb{R}^2 : x_1 \geq 0, x_2 \geq 0\}$.

2.2. Operations and details

Due to the fact that dominated solutions are not stored in our structure, our modified quad-tree actually reduces to a modified binary tree. Let Π be the set of nodes in the tree. For a given $\pi \in \Pi$, notice that if solutions are present in $\pi+\mathbb{R}_{++}$, they are dominated by π and should not be stored in the tree. Similarly, if solutions are present in $\pi+\mathbb{R}_{--}$, these solutions dominate π and π should be removed from the tree. Thus, for any node $\hat{\pi} \in \Pi$ the children of $\hat{\pi}$ associated with $\hat{\pi}+\mathbb{R}_{++}$ and $\hat{\pi}+\mathbb{R}_{--}$ are unnecessary. Hence, each $\hat{\pi} \in \Pi$ has only two children, and thus the tree reduces to a binary tree.

In order to present our structure in a clear, understandable manner, we define the following terms for each $\pi \in \Pi$: (1) $\pi.\texttt{type}$ – Sgmt if π represents a line segment, and Pnt if π represents a singleton, (2) $\pi.x_1, \pi.x_2, \pi.y_1$ and $\pi.y_2$ – π is identified by the point $(\pi.x_1, \pi.y_1)$ if $\pi.\texttt{type} =$ Pnt and the extreme points $(\pi.x_1, \pi.y_1)$ and $(\pi.x_2, \pi.y_2)$ if $\pi.\texttt{type} =$ Sgmt, (3) $\pi.p$ – parent node of π, (4) $\pi.l$ – left child node of π, (5) $\pi.r$ – right child node of π, (6) $\pi.\texttt{size}$ – total number of nodes contained in the subtree rooted at π, (7) $\pi.\texttt{ideal\_left} = (\pi^{nw}.x_1, \pi.y_1)$



(a)Local ideal points. (b)Partition $\mathbb{R}^2$ if $\pi.\texttt{type} = \mathsf{Pnt}$. (c)Partition $\mathbb{R}^2$ if $\pi.\texttt{type} = \mathsf{Sgmt}$.

FIGURE 2. Visuals for $\pi.\texttt{ideal\_right}$, $\pi.\texttt{ideal\_left}$, and the partitioning of $\mathbb{R}^2$ relative to π.

where π^{nw} is the north-west-most node in the subtree rooted at π, and (8) $\pi.\texttt{ideal\_right} = (\pi.x_2, \pi^{se}.y_2)$ where π^{se} is the south-east-most node in the subtree rooted at π. We say that π is the *root node* if $\pi.p = \emptyset$ and π is instead a *leaf node* if $\pi.l = \pi.r = \emptyset$ See Figure 2(a) for the details of $\pi.\texttt{ideal\_left}$ and $\pi.\texttt{ideal\_right}$.

Now, in order to further simplify the descriptions of the algorithms we use in implementing our data structure, we partition $\mathbb{R}^2$ into 4 regions relative to any node π. Figures 2(b) and 2(c) show the details of this partition for each type of node. We denote these regions by $R_\alpha(\pi)$ where $\alpha \in \{1,2,3,4\}$ represents the number of the region as shown in Figures 2(b) and 2(c). Given distinct nodes π and π^*, we use the notation $\pi^* \cap R_\alpha(\pi)$ to denote any portion of the point or segment associated with node π^* that lies in region $R_\alpha(\pi)$. If no such portion exists, we say $\pi^* \cap R_\alpha(\pi) = \emptyset$. To ensure that these regions are disjoint we assume that each region contains its lower and left boundaries, but not its upper or right boundaries. We also assume that π itself is contained in $R_2(\pi)$ and not $R_i(\pi)$ for $i \in \{1,3,4\}$. This convention is taken so that weakly dominated points will not be included in our structure.

This data structure has three main purposes: (i) it should be able to handle the insertion of several thousand solutions and update itself efficiently, (ii) the structure must be organized so that it can easily be searched and a desired subset can be obtained, and (iii) it must be able to return the current set of nondominated solutions. So, the main algorithms needed for the utilization of this data structure are functions for *insertion* of new solutions, *deletion* of dominated solutions, and *rebalancing* of the tree. We describe these algorithms next.

2.2.1. Insertion Recall that $\Omega_P \subseteq \cup_{i=1}^k S_i$ and is hence a collection of points and segments. Thus only points or segments will be inserted into the structure. For this purpose we define the INSERT function which takes two inputs: a node π^* which is being inserted and a node π which is the root of the tree or subtree where π^* is inserted. The point or segment associated with π^* is compared against π. Consider the following four situations:

(1) If $\pi^* \subseteq R_2(\pi)$ then $\pi \leqq \pi^*$ and thus π^* is discarded.
(2) If $\pi \not\leqq \pi^*$ but $\pi^* \cap R_2(\pi) \neq \emptyset$ then a portion of π^* is either dominated by π or is a repetition of solutions stored in π. We denote this situation by $\pi \leqq_p \pi^*$. In this case $\pi^* \cap R_2(\pi)$ is discarded.
(3) If $\pi^* \leqq \pi$ then π is removed from the tree.
(4) If $\pi^* \not\leqq \pi$ but $\pi \cap R_2(\pi^*) \neq \emptyset$ then π is reduced to $\pi \setminus R_2(\pi^*)$.

Note that the second possibility above may result in π^* being split into two disjoint pieces. Similarly, the final possibility may result in π being split into two nodes. If none of the above scenarios occur, neither π nor π^* dominates the other and they can coexist in the tree.

The typical use of the INSERT function is to insert a new node π^* at the root node, π_0. Then π^* is either discarded or $\pi^* \cap R_1(\pi)$ and $\pi^* \cap R_4(\pi)$ are inserted at $\pi.l$ and $\pi.r$, respectively. This process repeats recursively until either (i) π^* has been fully discarded, or

(ii) all nondominated portions of π^* have been added to the tree as new nodes. Note that π^* is added to the tree if and only if it is inserted at an empty node.Throughout the remainder of this paper we will use the notation REPLACE$(\pi', \tilde{\pi})$ to denote the process of replacing the point or segment associated with $\pi' \in \Pi$ with the point or segment associated with $\tilde{\pi} \in \Pi$ and leaving the tree structure otherwise unchanged. We use the notation $\pi' \leftarrow \tilde{\pi}$ to denote the process of replacing π' and its entire subtree with $\tilde{\pi}$ and its entire subtree. Algorithm 1 describes the INSERT procedure.

Algorithm 1 Inserting a new point or segment, π^*, into the data structure at node π

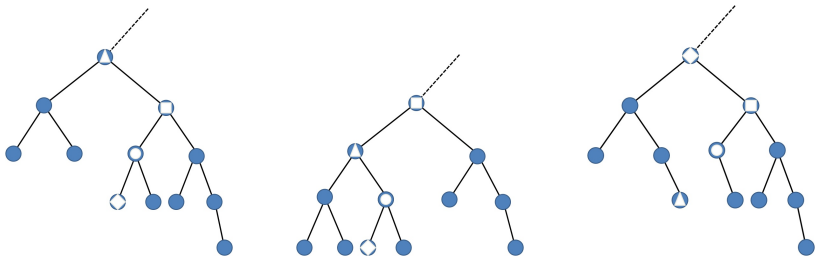
```
1: function INSERT(π*, π)
2:     if π* = ∅ then Return
3:     if π = π0 & π0 ≠ ∅ then REBALANCE(π)                ▷ π0 represents the root node
4:     if π = ∅ then REPLACE(π, π*), π.size ← 1, UPDATE(π)
5:     else REPLACE(π, π \ cl(R2(π*)))
6:         if π = ∅ then
7:             if π.ideal_left ∩ R2(π*) ≠ ∅ then π.l ← ∅
8:             if π.ideal_right ∩ R2(π*) ≠ ∅ then π.r ← ∅
9:             REMOVENODE(π)
10:            INSERT(π*, π)
11:        else
12:            if ∃ π1, π2 s.t. π = π1 ∪ π2 & cl(π1) ∩ cl(π2) = ∅ then
13:                π1.l ← π.l, π2.r ← π.r
14:                π ← π1, π.r ← π2
15:                UPDATE(π)
16:            INSERT(π* ∩ R1(π), π.l)
17:            INSERT(π* ∩ R4(π), π.r)
```

In Algorithm 1, the functions REMOVENODE and REBALANCE refer to the processes of deleting nodes from the tree and rebalancing the tree, respectively. These algorithms will be discussed further in Sections 2.2.2 and 2.2.3, respectively. The recursive UPDATE function has a node π as input and performs two actions: (i) ensures that $\pi.\texttt{size} = (\pi.l).\texttt{size} + (\pi.r).\texttt{size} + 1$ where $(\pi').\texttt{size} = 0$ if and only if $\pi' = \emptyset$, and (ii) ensures that $\pi.\texttt{ideal\_left}$ and $\pi.\texttt{ideal\_right}$ are updated appropriately. After this, if $\pi.p \neq \emptyset$ then UPDATE$(\pi.p)$ is called.

We now introduce a property that is maintained throughout all operations on the tree as described in the remainder of the paper.

Property 1. Given $\pi \in \Pi$, all nodes in the subtree of $\pi.l$ are located completely within $R_1(\pi)$ and all nodes in the subtree of $\pi.r$ are located completely within $R_4(\pi)$.

2.2.2. Deletion Removing a dominated node from the tree is the next task that frequently needs to be performed. Notice that when a node is deleted, in order for the tree structure to be retained, another node must replace it. This is precisely where the difficulty lies. Usually, when data is deleted from a quad-tree structure, all data contained in the subtree of the deleted node is reinserted in order to maintain proper organization of the tree [15, 16]. Since our quad-tree simplifies to a binary tree, however, we propose something much simpler. Notice that in order for our tree to maintain the appropriate structure, Property 1 must be met. For any node π that needs to be removed and replaced, there are precisely two nodes that may replace it and satisfy Property 1. They are the right-most node in the subtree of $\pi.l$ and left-most node in the subtree of $\pi.r$. For this task we define the REMOVENODE function, which is described in Algorithm 2.



(a)Original Tree (b)RebalanceLeft1 (c)RebalanceLeft2

FIGURE 3. Examples of applying rebalancing procedures.

Algorithm 2 Remove a node that has been shown to be dominated.

```
1:  function RemoveNode(π)
2:      if π.size = 1 then π → ∅
3:      else Define π̃ = ∅
4:          if (π.l).size > (π.r).size then
5:              π̃ → FindRightmostNode(π.l)
6:          else
7:              π̃ → FindLeftmostNode(π.r)
8:          Replace(π, π̃)
9:          RemoveNode(π̃)
10:     if π.p ≠ ∅ then Update(π.p)
```

Algorithm 3 Check to ensure that the balance criterion is met at each node.

```
1:  function Rebalance(π)
2:      if (π.l).size > 2 then Rebalance(π.l)
3:      if (π.r).size > 2 then Rebalance(π.r)
4:      if (π.l).size > π.size/(2−δ) then
5:          if (π.l.l).size ≥ (1−δ)π.size/(2−δ) − 1 then
6:              RebalanceRight1(π)
7:          else repeat
8:                  RebalanceRight2(π)
9:              until (π.l).size = π.size/(2−δ)
10:     else if (π.r).size > π.size/(2−δ) then
11:         if (π.r.r).size ≥ (1−δ)π.size/(2−δ) − 1 then
12:             RebalanceLeft1(π)
13:         else repeat
14:                 RebalanceLeft2(π)
15:             until (π.r).size = π.size/(2−δ)
```

2.2.3. Rebalancing The final task to perform in maintaining our structure is rebalancing. To maintain balance we use the following strategy of Overmars and Van Leeuwen [8]: for each non-leaf node π, the subtrees of $\pi.l$ and $\pi.r$ must contain no more than $\frac{1}{2-\delta}k$ nodes, where k is the number of nodes in π's subtree and δ is a preselected value in the open interval $(0, 1)$. Enforcing this requirement causes the depth of the tree to be at most $\log_{2-\delta} t$ where t is the number of nodes in the tree. Now, based on this requirement we develop two rebalancing methods, RebalanceLeft1 and RebalanceLeft2 (and similarly RebalanceRight1 and RebalanceRight2) each of which take a node π as input. In RebalanceLeft2, the left-most node of the subtree of $\pi.r$ is found and is used to replace π. Then π is moved to right-most position of the subtree of $\pi.l$. Notice that RebalanceLeft2 moves a single node from one side of a tree to the other. In certain situations it may be more beneficial to move several nodes from one side of the tree to the other in a single operation. RebalanceLeft1 is designed for this purpose. In RebalanceLeft1, the nodes of the tree are shifted in the following fashion: (i) $\pi.r$ and its right subtree shift up and left to take the place of π and its right subtree, (ii) π and its left subtree shift down and left to become the new left subtree of $\pi.r$, and (iii) the original left subtree of $\pi.r$ is then placed as the new right subtree of π. RebalanceLeft1 and RebalanceLeft2 are illustrated in Figure 3.

Algorithm 3 is used to determine which rebalancing procedure to apply in order to balance the tree. Its correctness is shown in Proposition 7, which is presented in the next section.

2.3. Performance Guarantees

We now present results about the correctness and complexity of the insertion, deletion, and rebalancing procedures. In this section we will use the notation $\overline{\pi} \in \text{Subtree}(\hat{\pi})$ to denote the case in which $\overline{\pi}$ is a node contained in the subtree of Π which is rooted at $\hat{\pi}$.

Proposition 1. INSERT *removes any portion of a currently stored node π which is dominated by an inserted node π^*.*

Proof Assume $\pi^* \leqq_p \pi \in \Pi$. Consider any $\hat{\pi}$ such that $\pi \in \text{Subtree}(\hat{\pi})$ and suppose π^* is being inserted at $\hat{\pi}$. Also, notice that the case in which $\pi = \pi_0$ is trivial, so we can assume WLOG that $\pi \in \text{Subtree}(\hat{\pi}.l)$. Suppose that $\pi^* \leqq \hat{\pi}.\texttt{ideal\_left}$. In this case $\text{Subtree}(\hat{\pi}.\texttt{ideal\_left})$ is removed from Π. Thus, since $\pi \in \text{Subtree}(\hat{\pi}.l)$, π is also removed from Π and the proposition is satisfied. Now suppose instead that $\pi^* \not\leqq \hat{\pi}.\texttt{ideal\_left}$. If $\pi^* \cap cl\,(R_2(\hat{\pi})) = \emptyset$, π^* will be inserted at $\hat{\pi}.l$ because otherwise $\pi^* \subset R_3(\hat{\pi})$ and thus $\pi^* \not\leqq_p \pi$ since $\pi \subset R_1(\hat{\pi})$. If, on the other hand, $\pi^* \cap cl\,(R_2(\hat{\pi})) \neq \emptyset$, there must exist $\pi^{**} \subset \pi^*$ such that $\pi^{**} \subset R_1(\hat{\pi})$ and $\pi \cap cl\,(R_2(\pi^{**}) = \pi \cap cl\,(R_2(\pi^*)$. If not, $\hat{\pi} \leqq_p \pi$ since $\pi^* \leqq_p \pi$, but this is a contradiction as it violates Property 1. Therefore π^{**} will be inserted at $\hat{\pi}.l$. Since these arguments hold for all $\hat{\pi} \in \Pi$ such that $\pi \in \text{Subtree}(\hat{\pi})$, one of the following must occur: (i) π is removed from Π due to a dominated ideal point, or (ii) there exists $\tilde{\pi} \subset \pi^*$ such that $\pi \cap cl\,(R_2(\tilde{\pi}) = \pi \cap cl\,(R_2(\pi^*)$ which will be inserted at π. During this insertion π will be reduced to $\pi \setminus cl\,(R_2(\pi^*)$. □

Proposition 2. INSERT *adds a portion of π^* to the tree if and only if it is not dominated by any node currently stored in the tree.*

Proof Notice that the reverse direction is trivial because if $\hat{\pi}$ is a portion of π^* not dominated by any $\pi \in \Pi$, then $\hat{\pi}$ will be inserted at one of the children of every node it is compared against. Thus, since there are a finite number of nodes in the tree, $\hat{\pi}$ must eventually be inserted at an empty node and added to the tree. For the forward direction we show the contrapositive. Suppose there is $\pi \in \Pi$ such that $\pi \leqq_p \pi^*$. Let $\tilde{\pi} = \pi^* \cap cl\,(R_2(\pi))$ and consider $\hat{\pi} \in \Pi$ such that $\pi \in \text{Subtree}(\hat{\pi})$. The case in which $\pi = \pi_0$ is trivial, so we can assume WLOG that $\pi \in \text{Subtree}(\hat{\pi}.l)$. Suppose that $\tilde{\pi} \subset cl\,(R_2(\hat{\pi}))$. In this case $\tilde{\pi}$ will not be included in any portion of π^* which is inserted to either child of $\hat{\pi}$. Thus the Proposition is satisfied. Now suppose instead that $\tilde{\pi} \not\subset cl\,(R_2(\hat{\pi}))$. In this case there must exist $\pi' \subset \tilde{\pi}$ such that $\pi' \subset R_1(\hat{\pi})$ because otherwise $\pi \not\leqq_p \tilde{\pi}$ since $\pi \subset R_1(\hat{\pi})$. Thus π' is included in the portion of π^* that will be inserted at $\hat{\pi}.l$. Since these arguments hold for all $\hat{\pi}$ such that $\pi \in \text{Subtree}(\hat{\pi})$, one of the following must occur: (i) $\tilde{\pi}$ is completely discarded during an insertion at some $\check{\pi} \in \Pi$ such that $\pi \in \text{Subtree}(\check{\pi})$, or (ii) there exists $\pi^{**} \subset \pi^*$ such that $\pi^{**} \cap \tilde{\pi} = \tilde{\pi} \setminus \cup_{i\,:\,\pi \in \text{Subtree}(\pi_i)} cl\,(R_2(\pi_i))$ and π^{**} is inserted at π. □

Proposition 3. *The worst case complexity of* INSERT *is $O(t)$, where t is the number of nodes currently in the tree.*

Proof Suppose that π^* is being inserted and $\exists\, \pi, \pi' \in \Pi$ such that: (i) $\pi > \pi'$, and (ii) π' is the only node in the subtree of π that is not dominated by π^*. WLOG assume $\pi.l > \pi'$. Then when π^* is compared with π the subtree of $\pi.l$ cannot be deleted. Thus π will be removed and replaced with another node $\tilde{\pi}$. This process will repeat until $\tilde{\pi} = \pi'$, which will be when π' is the only node left in π's subtree. Thus we can see that it is possible for π^* to be compared with every node in a subtree, implying $O(t)$ complexity. □

Proposition 4. *Use of the* REMOVENODE *procedure does not violate Property 1.*

Proof Notice that if $\pi = \pi.p.l$ then $\pi.p$ is completely within $R_4(\pi)$ and if $\pi = \pi.p.r$ then $\pi.p$ is completely in $R_1(\pi)$. Thus, if π^* is the right-most node in the subtree of $\pi'.l$ for some node π', then π^* is the unique node in the subtree of $\pi'.l$ for which all other nodes in the subtree of $\pi'.l$ are completely within $R_1(\pi^*)$. Similarly, if π^* is the left-most node in the

subtree of $\pi'.r$ for some node π', then π^* is the unique node in the subtree of $\pi'.r$ for which all other nodes in the subtree of $\pi'.r$ are completely within $R_4(\pi^*)$. Therefore, by replacing a deleted node π with either the right-most node in the subtree of $\pi.l$ or the left-most node in the subtree of $\pi.r$, Property 1 is satisfied even after the replacement. □

Proposition 5. *If the tree is perfectly balanced, the worst case complexity of* RemoveNode *is* $O(\log t)$*, where* t *is the number of nodes currently in the tree.*

Proof Recall that when a node π is removed it is replaced with either the left-most node in the subtree of $\pi.r$ or the right-most node in the subtree of $\pi.l$. Assuming that the tree is balanced, finding such a node is clearly an $O(\log t)$ process. If π' is the node replacing π and π' is not a leaf node, then its original position must then be filled using the same process. Note though that in finding the replacement for π', a path through the tree is traversed which begins precisely where the path traversed in finding the replacement for π ended. Thus, even though multiple nodes may need replaced in order for π to be removed, the overall process must result in the traversal of only one path through the tree, resulting in an $O(\log t)$ procedure. □

Proposition 6. *Use of the* Rebalance *procedure does not violate Property 1.*

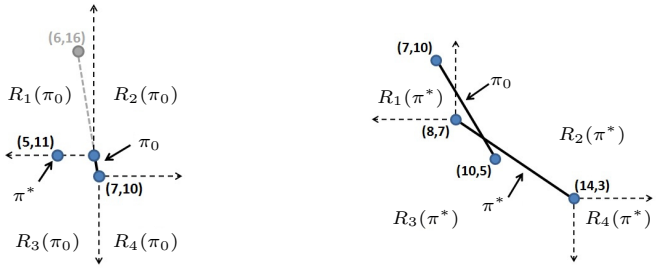
Proof To show that the proposition holds, we must show that neither RebalanceLeft1 nor RebalanceLeft2 violates Property 1. First consider RebalanceLeft1(π). Note that after this procedure is carried out, $\pi.r$ becomes the root node of the subtree that was once rooted at π. All nodes that were in the subtree of $\pi.r.l$ remain in their original positions relative to $\pi.r$. Now notice that π becomes the left child of $\pi.r$, which does not violate Property 1 since π is completely within $R_1(\pi.r)$. Finally, the entire subtree of $\pi.r.l$ becomes the right subtree of π. Since π is now the left child of $\pi.r$, all of these nodes are still located in the left subtree of $\pi.r$. Furthermore, since these nodes were originally located in π's right subtree, Property 1 is still satisfied. Now consider RebalanceLeft2(π). In this procedure π is replaced by the left-most node in the subtree of $\pi.r$. We proved that this would not violate Property 1 in the proof of Proposition 4. After this, π is placed as the right child of the node that was previously the right-most node in the subtree of $\pi.l$. This placement also does not violate Property 1 since all nodes originally within the subtree of $\pi.l$ are completely within $R_1(\pi)$. □

Proposition 7. *After one call of* Rebalance(π) *the balance criterion is satisfied at* π*.*

Proof WLOG assume that $(\pi.r).\texttt{size} > \frac{\pi.\texttt{size}}{2-\delta}$. Now, if $(\pi.r.r).\texttt{size} < \frac{(1-\delta)\pi.\texttt{size}}{2-\delta} - 1$ then the proposition is trivially satisfied since in this case RebalanceLeft2 is repeated until $(\pi.r).\texttt{size} = \frac{\pi.\texttt{size}}{2-\delta}$. Thus, we focus on the case in which $(\pi.r.r).\texttt{size} \geq \frac{(1-\delta)\pi.\texttt{size}}{2-\delta} - 1$. Notice that by the construction of the Rebalance procedure, the subtrees of $\pi.l$ and $\pi.r$ are balanced before that of π. Thus $(\pi.r.r).\texttt{size} \leq \frac{\pi.\texttt{size}}{2-\delta}$ because otherwise $(\pi.r.r).\texttt{size} > \frac{\pi.\texttt{size}}{2-\delta} > \frac{(\pi.r).\texttt{size}}{2-\delta}$ which contradicts the fact that the subtree of $\pi.r$ is balanced. Now, suppose that after calling RebalanceLeft1(π), π' is the new root node of the subtree orignally rooted at π. Then the subtree of $\pi'.r$ will be the original subtree of $\pi.r.r$. Thus, since $(\pi.r.r).\texttt{size} \leq \frac{\pi.\texttt{size}}{2-\delta}$, the balance criterion will be satisfied for $\pi'.r$. Also notice that RebalanceLeft1(π) is only called if $(\pi.r.r).\texttt{size} \geq \frac{(1-\delta)\pi.\texttt{size}}{2-\delta} - 1 \Rightarrow (\pi.r.r).\texttt{size} \geq \frac{(1-\delta)\pi.\texttt{size}+\pi.\texttt{size}-\pi.\texttt{size}}{2-\delta} - 1 \Rightarrow \frac{\pi.\texttt{size}}{2-\delta} \geq \pi.\texttt{size} - (\pi.r.r).\texttt{size} - 1$. After the procedure is completed, it will be the case that $(\pi'.l).\texttt{size} = \pi.\texttt{size} - (\pi.r.r).\texttt{size} - 1$ where $\pi.\texttt{size}$ is the size of the original subtree rooted at π. Thus, the balance criterion will be satisfied for $\pi'.l$. □

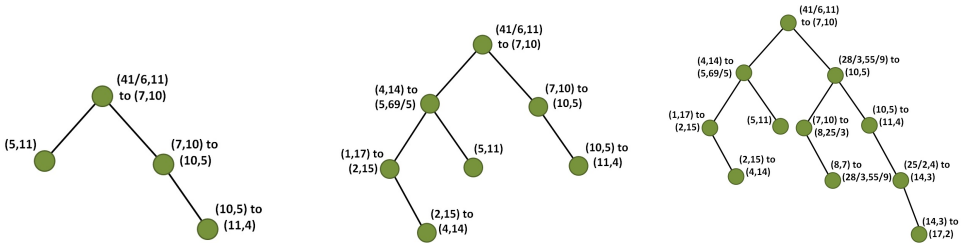
Proposition 8. *The worst-case complexity of maintaining a balanced tree is* $O(t^2 \log t)$*.*

Proof As currently implemented, rebalancing requires checking the balance criterion at every node of the tree. Ensuring that the balance criterion is met at one of these nodes



(a)Inserting (5,11). (b)Inserting $(8,7)$-$(14,3)$.

FIGURE 4.



(a)Tree after inserting (5,11). (b)Tree after rebalancing. (c)Final tree.

FIGURE 5.

could require repeating the strategy REBALANCELEFT2 up to $\frac{t}{2}$ times. Thus, since REBALANCELEFT2 calls FINDLEFTMOSTNODE, FINDRIGHTMOSTNODE, and UPDATE which are $O(\log t)$ procedures, the complexity of rebalancing is $O(t^2 \log t)$. □

3. Illustrative example

Recall the points and segments specified in Figure 1(a). We use these points and segments as input to our data structure and show a few of the nontrivial steps of developing our tree. Assume that the solutions shown in the figure are obtained from five separate slice problems and that the Pareto sets of these slice problems, listed in respective order, are: (i) the singleton (1,19), (ii) the piecewise linear curve connecting (1,17) and (9,13), (iii) the piecewise linear curve connecting (6,16) and (11,4), (iv) the singleton (5,11), and (v) the piecewise linear curve connecting (8,7) and (17,2). The points and segments which define these Pareto sets will be inserted into our structure in the order of (iii), (iv), (ii), (v), (i). Piecewise linear curves will be inserted as individual line segments from left to right.

The reader is encouraged to review the pseudocode given previously (particularly Algorithm 1). To begin we let $\pi^* \leftarrow (6,16)$ to $(7,10)$ and call INSERT(π^*, π_0). Since $\pi_0 = \emptyset$ we replace π_0 with π^*. Clearly the current tree structure is now a single node. Next we let $\pi^* \leftarrow (7,10)$ to $(10,5)$ and call INSERT(π^*, π_0). Notice that $\pi^* \subset R_4(\pi_0)$ and should be inserted at $\pi_0.r$. Since $\pi_0.r = \emptyset$ this insertion results in π^* being added to the tree. Therefore the tree now contains the root node which has one child to its right. The insertion of the segment connecting $(10,5)$ to $(11,4)$ is analogous. Next consider Pareto set (iv). Let $\pi^* \leftarrow (5,11)$ and call INSERT(π^*, π_0). Observe Figure 4(a). Clearly we can see that $\pi^* \leq_p \pi_0$ and thus we remove the dominated portion of π_0 by letting $\pi_0 = \pi_0 \setminus R_2(\pi^*)$. After this has been done, notice that $\pi^* \subset R_1(\pi_0)$. Therefore, since $\pi_0.l = \emptyset$, π^* becomes the left child of π_0. Figure 5(a) shows the tree structure after π^* has been inserted. We leave it to the reader to consider Pareto set (ii). Note, though, that after processing this set the subtree rooted at $\pi_0.l$ needs to be rebalanced. The resulting tree is shown in Figure 5(b).

Next we consider the insertion of Pareto set (v). Let $\pi^* \leftarrow (8,7)$ to $(14,3)$ and call INSERT(π^*, π_0). Clearly $\pi^* \subset R_4(\pi_0)$ and will therefore be inserted to $\pi_0.r$. Observe from Figure 4(b) that $\pi^* \leqq_p \pi_0.r$. This time, though, the portion of $\pi_0.r$ which is dominated is the center section of the segment. This means that $\pi_0.r$ must be split into two nodes π_1 and π_2. Node π_1 takes the place in the tree where $\pi_0.r$ originally was, and the left subtree of $\pi_0.r$ becomes the left subtree of π_1. Node π_2 becomes the right child of π_1 and the right subtree of $\pi_0.r$ becomes the right subtree of π_2. Now, after this process has been completed, observe that $\pi^* \subset R_4(\pi_1)$ and thus π^* will be inserted to π_2 (which is now $\pi_0.r.r$). Notice that $\pi_0.r.r \leqq_p \pi^*$ and that it is the center portion of π^* that is dominated. Thus the calls to INSERT$(\pi^* \cap R_1(\pi_0.r.r), \pi_0.r.r.l)$ and INSERT$(\pi^* \cap R_4(\pi_0.r.r), \pi_0.r.r.r)$ will each cause a portion of π^* to be inserted at $\pi_0.r.r.l$ and $\pi_0.r.r.r$ respectively. Since $\pi_0.r.r.l = \emptyset$, $\pi^* \cap R_4(\pi_0.r.r)$ will become $\pi.r.r.l$. Since $\pi_0.r.r.r$ is the segment (10,5) to (11,4), it is clear that another portion of π^* will need to be removed, and then the remainder of π^* will become $\pi_0.r.r.r.r$.

We end our example now because the remainder of the insertions result in scenarios which are analogous to those that we have now observed. Note that if we were to continue, one more rebalance would be required and the final tree structure would be that found in in Figure 5(c). Note that this tree structure is dependent on the order of insertion.

4. Computational Experiments

We implemented our data structure in the C programming language and performed two tests. The first was designed to test the amount of data our structure can effectively store and how quickly it can be processed. The second was designed to test the utility of our data structure when used alongside the BB algorithm of Belotti et al. [1]. Both tests were run using Clemson University's Palmetto Cluster. Specifically, an HP SL250s server node with a single Intel E5-2665 CPU core with 16GB of RAM running Scientific Linux 6.4 was used.

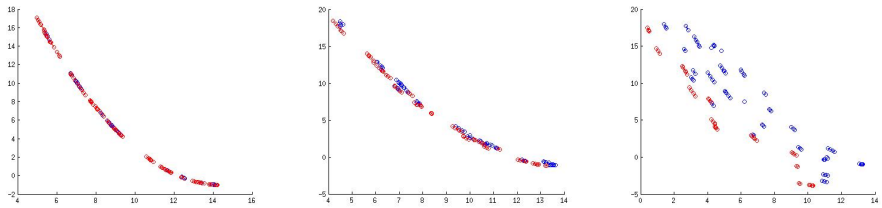
4.1. Implementation of Rebalance

Recall from Propositions 3, 5, and 8 that maintaining a balanced tree is the most costly of the three operations needed to create our structure. It is also the one operation that is unnecessary in order to ensure that we store the correct solutions. For this reason we decided to further consider the rebalancing operations in hopes of finding an alternative implementation that is less computationally costly, but still performs well in practice. We denote our initial implementation as A1, and develop an alternate implementation A2 in which rebalance is performed periodically rather than before every insertion at the root node. We check the entire tree for balance after 100 new solutions are added to the tree and then again each time the number of stored nodes doubled. We implemented both approaches in our first experiment, which is described in Section 4.2. We utilize approach A2 when performing our second experiment, described in Section 4.3, since it provided faster running times in our first experiment.

4.2. Experiment 1 - Random Data

4.2.1. Setup This test has two main purposes: (i) to compare the efficiency of our data structure with that of a dynamic list (which updates via pairwise comparison) when storing nondominated solutions, and (ii) to determine how many solutions our structure can take as input and process in a reasonable amount of time.

The test consists of repeating the following procedure until N insertions have been made into our structure or the dynamic list. First, generate a random integer $i \in [1,6]$ and a random number $r_1 \in (0,10)$. Then, if $i > 1$, for each $j \in \{2,\dots,i\}$ a random number $c_j \in (0,1)$ is generated and we define $r_j = r_1 + \sum_{\ell=2}^{j} c_\ell$. Next, for each $j \in \{1,\dots,i\}$ the following are computed: (i) $y_j = \frac{(10.5-r_j)^2}{5} - k$, and (ii) $x_j = r_j + (5-k)$. Here k is a dynamic value which is defined as 1 at the start of the test and increases by $\frac{\mu}{N}$ each time the above process is



(a)$\mu = 0.1$ (b)$\mu = 1$ (c)$\mu = 10$

FIGURE 6. Example of solutions generated in Experiment 1 with $N = 100$.

repeated. Here $\mu \in \mathbb{R}$ is a parameter that allows us to determine how much the solutions should "improve" over the course of the test. If $i = 1$, the singleton (x_1, y_1) is inserted into the structure, otherwise the points $(x_1, y_1), ..., (x_i, y_i)$ are arranged in order of increasing x values and then the line segments connecting each adjacent pair of points are inserted into the structure. We performed this test 100 times for each combination of the values $N = 10^4, 10^5, 10^6$ and 10^7 and $\mu = 0, 0.001, 0.01, 0.1, 1$ and 10. We used various values for δ and found that the results were quite similar, but determined to use a value of $\delta = 0.3$. For each test we recorded the time it took to insert all solutions into our structure and a dynamic list, and the final number of nodes stored in our tree and the dynamic list.

We now explain the significance of μ. Selecting a small value of μ (close to 0) is intended to replicate instances of BOMILP in which there is little separation between integer feasible solutions (i.e., all solutions are close to being Pareto optimal). Alternatively, selecting a large value of μ is intended to replicate instances of BOMILP in which there is a lot of separation between integer feasible solution. Figure 6 shows an example of solutions generated during this experiment for $\mu = 0.1, 1$ and 10 and for $N = 100$. The solutions shown in red are those that are stored by our structure at the end of the test.

4.2.2. Implementation Details First, recall that as presented, the implementation of our structure performs a check in order to determine whether or not an entire subtree is dominated. If a subtree is found to be dominated, the entire subtree is removed. We found that in practice, however, this implementation does not outperform the implementation in which no check for dominated subtrees is performed, rather dominated nodes are removed one at a time. We feel that there are two drawbacks to the former implementation which are most likely the reasons for this: (i) more information (i.e, an ideal point for each subtree) is stored in each node, and (ii) when new solutions are added to the tree, the UPDATE function must ensure that these ideal points are updated appropriately, which can be a costly procedure. Also notice that the worst case complexity of REMOVENODE remains the same for both implementations. For this reason, we used the latter implementation when performing our tests.

4.2.3. Numerical Results We present the results obtained from our randomized tests when implementing the rebalancing approaches discussed in section 4.1.2. Our results are summarized in Table 1. For each pair of values for μ and N the minumum, maximum, and average time to process all inserted solutions was recorded for our tree using approaches A1 and A2 and for a dynamic list (L). The average number of stored nodes was also recorded for each. Note that we terminated individual runs that were not completed in 12 hours and this is why certain data is missing in the Table. Furthermore, if an individual run took more than 8 hours to complete, only 5 replications were performed instead of 100. This situation is indicated in Table 1 by the symbol ⊛.

There are several things to notice from Table 1. First, in all cases our data structure is able to process inserted solutions much more quickly than the dynamic list. Furthermore,

TABLE 1. Avgerage time to process input and number of nodes stored for Experiment 1.

		Rebal	Time (s)			Number of			Rebal	Time (s)			Number of
μ	N	Type	Min	Avg	Max	Stored Nodes	μ	N	Type	Min	Avg	Max	Stored Nodes
0	10^4	A1	1.76	2.02	2.89	25,730	0.01	10^4	A1	0.15	0.16	0.17	767
		A2	0.29	0.36	0.69	25,758			A2	0.04	0.05	0.05	767
		L	243	416	1,156	25,170			L	0.62	0.72	0.77	766
	10^5	A1	479	496	639	198,158		10^5	A1	3.57	3.82	4.48	2,369
		A2	12.9	17.0	42.0	199,361			A2	0.73	0.76	0.81	2,369
		L	–	–	–	–			L	30.4	34.0	38.1	2,365
	10^6	A1	–	–	–	–		10^6	A1	68.9	102	132	7,209
		A2	353	398	558	868,442			A2	10.8	14.6	16.0	7,208
		L	–	–	–	–			L	2,078	2,405	5,596	7,171
	10^7	A1	–	–	–	–		10^7	A1	2,008	2,259	3,160	22,413
		A2	3,696	4,589	7,304	2,170,820			A2	213	258	338	22,338
		L	–	–	–	–			L	–	–	–	–
0.1	10^4	A1	0.05	0.06	0.06	285	10	10^4	A1	0.03	0.03	0.03	133
		A2	0.02	0.02	0.03	286			A2	0.01	0.02	0.02	133
		L	0.13	0.15	0.19	285			L	0.06	0.06	0.06	133
	10^5	A1	1.67	1.69	1.73	766		10^5	A1	0.45	0.46	0.47	
		A2	0.41	0.42	0.43	767			A2	0.21	0.22	0.22	189
		L	7.49	7.68	7.86	765			L	0.92	0.92	0.93	189
	10^6	A1	35.7	38.2	41.3	2,366		10^6	A1	6.36	6.41	6.62	284
		A2	6.66	6.79	6.95	2,367			A2	2.48	2.51	2.60	285
		L	306	347	401	2,363			L	14.3	14.4	14.5	284
	10^7	A1	626	653	703	7,212		10^7	A1	139	140	145	764
		A2	92.7	98.9	116	7,209			A2	36.5	36.6	37.6	764
		L⊛	18,977	19,548	20,446	7,166			L	528	529	543	764

processing time is reduced significantly when rebalancing approach A2 is used instead of A1. Also notice that for each fixed value of N, the time taken to process inserted solutions decreases as the value of μ increases. Additionally, the larger the value of μ, the closer the time needed for the dynamic list to process the input solutions becomes to the time needed for our tree to process the solutions. From these results we can see that our data structure can handle the insertion of large sets of solutions, thus we suspect that it can do so without posing a significant overhead on a solution procedure such as BB or a heuristic method.

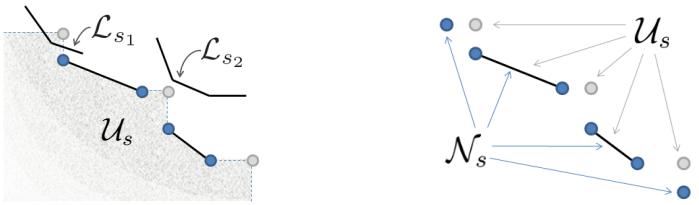
4.3. Experiment 2 - Fathoming in BB

4.3.1. Setup We performed tests in which we solved a variety of instances of BOMILP using the BB technique of Belotti et al. [1]. Each instance was solved three times, once using our structure in order to generate the upper bound set at each iteration of the BB, once using a dynamic list in order to generate these sets, and once using a predetermined subset of Ω_P to generate a single upper bound set which was used for fathoming throughout the BB. More details on each of these types of sets are provided in the following section.

The instances of BOMILP that we solved were taken from [1] and [2]. We present results on all instances which took over 10 seconds, but under 8 hours to solve. In the following section we provide a brief background on biobjective BB procedures.

4.3.2. Background on BB Many of the prevalent techniques for solving BOMILP are based on the branch-and-bound (BB) method, which has been well established in the single-objective case. For an extensive survey of single-objective BB we suggest [18]. The efficiency of single-objective BB methods heavily relies on the fact that upper and lower bounds for the optimal solution can often be determined. Similarly, in the biobjective case subsets of $\mathbb{R}^2$ can often be found which bound the Pareto set above and below. In general these sets are formed by taking unions of finitely many continuous convex piecewise linear functions. More detailed discussion on these sets can be found in [1, 4].

During an iteration of a typical BB procedure one solves an LP subproblem at a node η selected from a list $\mathscr{L}$ of open subproblems in the BB enumeration tree. Under certain conditions the upper and lower bound sets can be used to prove that a particular subproblem cannot yeild any Pareto solutions. In this case the node η associated with this subproblem can be cut off or *fathomed* from the BB enumeration tree. One of the fathoming rules presented by Belotti et al. [1] states that at iteration s of BB a node η_s can be fathomed if



(a)η_2 can be fathomed. (b)Examples of $\mathcal{N}_s$ and $\mathcal{U}_s$.

FIGURE 7. Examples of bound sets and fathoming rules.

the lower bound set $\mathcal{L}_s$ is separable from the upper bound set $\mathcal{U}_s$, i.e., $\mathcal{L}_\eta \cap (\mathcal{U}_s - \mathbb{R}^2_+) = \emptyset$. Figure 7(a) shows examples of lower bound sets $\mathcal{L}_{s_1}$ and $\mathcal{L}_{s_2}$. Notice that the locations of these sets show that η_{s_1} cannot be fathomed but η_{s_2} can. Clearly, efficient fathoming depends on the choice of $\mathcal{N}_s$ used to construct $\mathcal{U}_s$ since good approximations of $\mathcal{U}_G$ at each iteration of BB can help fathom a large number of nodes.

At iteration s of BB, let $\mathcal{F}_s$ be the set of all $\omega \in \Omega$ discovered during iterations $1, \ldots, s-1$. Then at iteration s, the upper bound set $\mathcal{U}_s$ is built using another set $\mathcal{N}_s$, the nondominated subset of $\mathcal{F}_s$. An example of constructing $\mathcal{U}_s$ from $\mathcal{N}_s$ can be seen in Figure 7(b). Finding $\mathcal{N}_s$ can be cumbersome and until now, there seem to have been only two approaches used:

Dynamic List: Each time $\omega \in \Omega$ is found, store it in a list and then remove dominated points and segments by performing a pairwise comparison between all stored solutions. After completion of the pairwise comparison the stored solutions are precisely $\mathcal{N}_s$ [7, 17].

Predetermined subset of Ω_P: Before beginning BB a preprocessing phase is used to generate a set $\mathcal{N} \subset \Omega_P$. Then at every iteration s of BB, let $\mathcal{N}_s = \mathcal{N}$. Therefore a single upper bound set is used for fathoming throughout the BB [1].

4.3.3. Implementation Details First we point out that when utilizing the predetermined subset of Ω_P, the ϵ-constraint method was used to generate $M \leq M^*$ points from Ω_P before beginning the BB, where M^* is a user-selected upper bound on the number of these points that are generated. Notice, though, that these M points can still be useful in the cases when either our structure or a dynamic list is being used alongside the BB. By inserting these points into either structure at the start of the BB, the procedure can be "warm-started," increasing the frequency and efficiency of fathoming.

Initially we solved several instances using our structure both with and without warm-starting. However, the results we obtained without warm-starting were very poor, and are therefore not reported. Notice that warm-starting allows solutions which are "far" from the set of Pareto-optimal solutions to be discarded early in the BB, and therefore fewer nodes of the BB tree are explored. We solved each instance using various values for M^*, ranging from 10 on small instances to 3000 on large ones. "Good" choices for the value of M^* seem to be highly dependent on the size and difficulty of the instance being solved.

4.3.4. Numerical Results The results we obtained from Experiment 2 are summarized in the performance profile of CPU time shown in Figure 8. We use P, T, and L to represent the implementations of the Predetermined set of Ω_P, our tree structure, and the dynamic list, respectively. From [1], there were 30 instances available for each problem size and from [2], there were 5 instances available for each problem size. However, we were unable to solve instances from [2] which had 320 variables and constraints, because they took longer than 8 hours to solve.

As it is difficult to know ahead of time what value of M^* is most appropriate for solving a given BOMILP, we ran each of these instances for a large number of values for M^*, but for the sake of space only report results for 3 values of M^* for each problem size. In Figure 8 we use L, M, and H to denote low, medium, and high values for M^*, respectively.

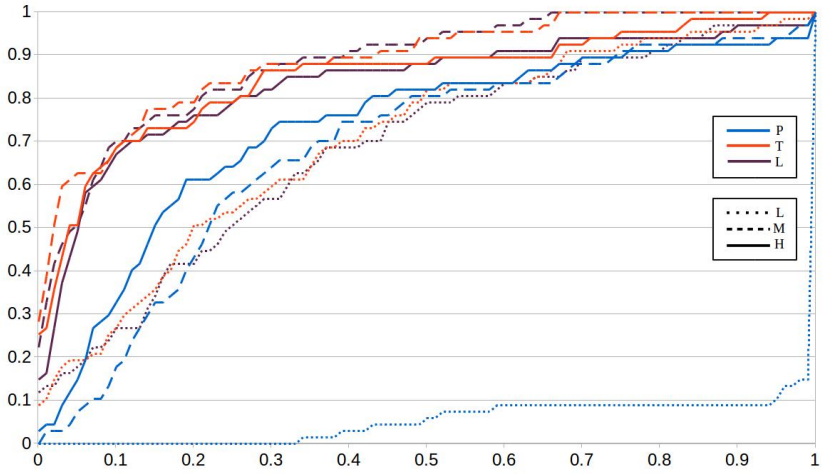


FIGURE 8. Performance Profile for solving BOMILP instances from [1, 2].

The profile shown in Figure 8 shows that our structure either outperforms or performs equally as well as both of the other implementations in almost all cases. We point out that the BB implemented using the predetermined subset of Ω_P for generation of the bound set does not have the Pareto set readily available upon termination. Instead, the set of all integer feasible solutions is stored and a post-processing phase is needed in order to determine the Pareto set. The implementations using our tree and the list data structure, on the other hand, do have the Pareto set readily available upon termination.

When comparing the results of this experiment to those of our first experiment one may wonder why our tree structure does not significantly outperform the list in all cases. On inspecting the number of solutions inserted to our structure versus the final number stored, we found that a value of $\mu \approx 100$ (cf. §4.3.1 and Figure 6) could be associated with most of the solved instances. This value of μ indicates that there is a high level of separation amongst the solutions generated during BB and therefore a large fraction of generated solutions ends up being dominated and hence not stored. Thus for BB experiments, there is not a significant difference between our structure and the list in terms of the time needed to process the data. In fact, the time for either structure to process all inserted solutions was approximately one order of magnitude less than the time for BB to solve the instance. Thus, the BB had a much larger impact on overall running time than the data structure used.

5. Conclusion

In this work we have introduced a new data structure, in the form of a modified binary tree, that is able to efficiently store sets of nondominated solutions of BOMILPs. Until now similar structures have only been used in the pure integer case. We provide an extension for the more difficult mixed-integer case. We showed this structure performs with a worst case guarantee of $O(t^2 \log t)$ where t is the number of stored nodes. We tested the practical value of our data structure with two experiments. The results show that our structure provides a more efficient method for storing solutions to BOMILP than other current techniques. They also show that our structure is also a very useful tool when used alongside BB methods for solving BOMILPs.

We recognize that there may be ways to extend our data structure and increase its efficiency. Recall that each node of our structure may store either a point or a line segment. It is possible that in certain cases our structure stores several segments that all belong to a single piecewise linear curve. Therefore it may be beneficial to extend the functionality of

our structure so that entire piecewise linear curves can be stored in a single node. Notice that in some cases this may allow for a significant reduction of the size of the tree and thus allow the structure to be populated and maintained more quickly. The reason that we did not implement our structure in this fashion is that for the BOMILP solution techniques we are familiar with, segments are generated one at a time and in general connecting segments are not generated sequentially. Also, for the specific instances we solved, it was not often that a significant number of connected line segments generated from the same slice problem were Pareto optimal.

Appendix

TABLE 2. Timing, fathoming, and storage results for Experiment 2 (reported as geometric averages). The first set of instances is taken from [1] and the second is taken from [2]. Problem sizes are reported as number of variables, which in all instances also equalled the number of constraints.

		Time (s)			Nodes Fathomed			# of Insertions		Final # of Nodes Stored		Final Depth
Size	M	P	T	L	P	T	L	T	L	T	L	of Tree
60	10	25.0	15.7	15.6	516	342	342	2,191	2,190	75.0	76.3	12.8
	25	14.7	12.7	12.7	309	271	271	1,421	1,421	75.8	78.4	13.4
	50	13.9	12.8	12.8	273	254	254	1,246	1,246	76.0	79.6	13.4
80	10	54.2	35.0	35.4	705	492	492	2,743	2,743	89.0	90.1	9.6
	25	37.1	31.2	31.4	488	428	428	2,102	2,102	89.5	91.6	10.1
	50	34.2	31.4	31.6	430	403	403	1,869	1,869	89.6	93.3	9.7
80	50	252	73.1	75.5	5,326	1,820	1,830	67,406	69,343	1,062	1,061	19
		158	77.4	76.9	2,758	1,397	1,397	120,402	117,626	673	694	12
		236	84.7	87.0	5,474	2,130	2,135	64,902	64,721	933	954	11
		189	57.1	58.9	4,058	1,343	1,350	58,425	58,405	928	947	11
		125	64.5	69.4	3,645	2,053	2,072	47,867	54,764	746	759	13
	200	63.3	47.6	50.2	1,357	1,039	1,040	37,601	37,561	1,052	1,104	15
		45.7	34.1	34.7	936	707	705	38,918	38,777	676	720	17
		74.3	50.6	51.8	1,608	1,211	1,215	27,527	26,243	916	992	12
		64.5	38.8	41.0	1,309	861	861	36,857	36,829	930	990	15
		61.7	45.6	46.9	1,763	1,351	1,352	30,833	30,880	757	799	14
	300	56.9	43.1	47.6	1,151	986	992	34,527	35,430	1,058	1,132	16
		41.3	35.8	36.8	836	700	701	42,148	42,105	676	718	12
		65.9	48.5	51.1	1,445	1,143	1,145	24,157	24,231	927	1,031	13
		59.5	38.1	40	1,172	806	804	35,296	35,067	938	1,012	13
		52.4	43.3	44.3	1,493	1,272	1,272	28,251	28,251	758	823	11
160	500	14,886	13,115	13,244	74,328	61,602	61,618	3,274,710	3,277,481	2,794	2,922	52
		24,763	21,427	21,217	154,107	119,310	119,341	2,806,323	2,816,796	2,976	3,079	39
		16,390	14,166	15,200	105,371	93,544	93,605	1,857,632	1,864,128	2,725	2,859	17
		8,833	6,983	7,324	43,776	36,310	36,385	1,319,040	1,344,686	6,156	6,310	26
		3,709	3,520	3,752	19,613	16,879	16,898	572,002	575,330	3,043	3,130	23
	2,000	10,617	10,359	10,590	53,375	50,999	51,014	2,484,279	2,486,110	2,802	3,280	28
		15,454	16,205	16,064	100,840	94,739	94,759	2,156,450	2,165,352	2,994	3,354	30
		13,023	13,233	13,487	88,858	86,171	86,459	1,449,196	1,475,056	2,765	3,214	33
		6,102	5,905	6,064	34,129	32,065	32,229	958,695	1,006,657	6,159	6,783	18
		2,957	2,846	2,880	14,945	14,110	14,133	461,307	462,446	3,083	3,410	13
	3,000	9,429	10,169	10,383	51,686	50,056	50,066	2,417,654	2,418,904	2,796	3,510	24
		14,946	15,683	15,978	96,730	92,126	92,225	2,090,566	2,118,231	2,982	3,519	24
		12,942	12,437	12,491	87,014	84,997	85,129	1,396,941	1,423,354	2,757	3,432	24
		5,978	5,965	5,951	33,273	31,569	31,636	916,966	941,870	6,179	7,110	22
		3,015	3,038	3,273	14,446	13,804	13,813	449,401	451,670	3,089	3,560	28

References

[1] Pietro Belotti, Banu Soylu, and Margaret M Wiecek. A branch-and-bound algorithm for biobjective mixed-integer programs. Technical report, Clemson University, December 2012. URL `http://www.clemson.edu/ces/math/technical_reports/belotti.bb-bicriteria.pdf`.

[2] Natashia Boland, Hadi Charkhgard, and Martin Savelsbergh. The triangle splitting method for biobjective mixed integer programming. In Jon Lee and Jens Vygen, editors, *Integer Programming and Combinatorial Optimization (IPCO)*, volume 8494 of *Lecture Notes in Computer Science*, pages 162–173. Springer International Publishing, 2014. doi: http://dx.doi.org/10.1007/978-3-319-07557-0_14.

[3] Jesús A De Loera, Raymond Hemmecke, and Matthias Köppe. Pareto optima of multicriteria integer linear programs. *INFORMS Journal on Computing*, 21(1):39–48, 2009.

[4] Matthias Ehrgott and Xavier Gandibleux. Bound sets for biobjective combinatorial optimization problems. *Computers & Operations Research*, 34(9):2674–2694, 2007.

[5] Nicolas Jozefowiez, Gilbert Laporte, and Frédéric Semet. A generic branch-and-cut algorithm for multiobjective optimization problems: Application to the multilabel traveling salesman problem. *INFORMS Journal on Computing*, 24(4):554–564, 2012.

[6] Gülseren Kiziltan and Erkut Yucaoğlu. An algorithm for multiobjective zero-one linear programming. *Management Science*, 29(12):1444–1453, 1983.

[7] George Mavrotas and Danae Diakoulaki. Multi-criteria branch and bound: A vector maximization algorithm for mixed 0-1 multiple objective linear programming. *Applied mathematics and computation*, 171(1):53–71, 2005.

[8] Mark H Overmars and Jan Van Leeuwen. Dynamic multi-dimensional data structures based on quad-and k?d trees. *Acta Informatica*, 17(3):267–285, 1982.

[9] Özgür Özpeynirci and Murat Köksalan. An exact algorithm for finding extreme supported nondominated points of multiobjective mixed integer programs. *Management Science*, 56(12):2302–2315, 2010.

[10] Anthony Przybylski, Xavier Gandibleux, and Matthias Ehrgott. A two phase method for multi-objective integer programming and its application to the assignment problem with three objectives. *Discrete Optimization*, 7(3):149 – 165, 2010. ISSN 1572-5286. doi: http://dx.doi.org/10.1016/j.disopt.2010.03.005.

[11] Ted K Ralphs, Matthew J Saltzman, and Margaret M Wiecek. An improved algorithm for solving biobjective integer programs. *Annals of Operations Research*, 147(1):43–70, 2006.

[12] Hanan Samet. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.

[13] Francis Sourd and Olivier Spanjaard. A multiobjective branch-and-bound framework: Application to the biobjective spanning tree problem. *INFORMS Journal on Computing*, 20(3):472–484, 2008.

[14] Thomas Stidsen, Kim Allan Andersen, and Bernd Dammann. A branch and bound algorithm for a class of biobjective mixed integer programs. *Management Science*, 60 (4):1009–1032, 2014.

[15] Minghe Sun. A primogenitary linked quad tree data structure and its application to discrete multiple criteria optimization. *Annals of Operations Research*, 147(1):87–107, 2006.

[16] Minghe Sun and Ralph E Steuer. Quad-trees and linear lists for identifying nondominated criterion vectors. *INFORMS Journal on Computing*, 8(4):367–375, 1996.

[17] Thomas Vincent, Florian Seipp, Stefan Ruzika, Anthony Przybylski, and Xavier Gandibleux. Multiple objective branch and bound for mixed 0-1 linear programming: Corrections and improvements for the biobjective case. *Computers & Operations Research*, 40(1):498 – 509, 2013.

[18] Laurence A Wolsey. *Integer programming*, volume 42. Wiley New York, 1998.